OutputBuilder Detailed Design Document

Architectural Overview - Web App and Worker

The OutputBuilder (pdfcombinercco) service consists of two Heroku processes:

- a Heroku Web App
- a Heroku Worker

The Heroku Web App runs on a web dyno and the Heroku Worker runs on a worker dyno, hence they are physically independent processes (and the Worker may not even be running when the Web App is running, or vice versa).

Communication between the two processes is achieved using a single queue on CloudAMQP and is therefore asynchronous. The Web App serves as the producer of messages on the queue, while the Worker serves as the consumer of messages on the queue. The Web App can publish messages to the queue even if the Worker is offline and when the Worker comes online, the messages will be consumed.

The Worker may be intentionally offline because a worker dyno is not free on Heroku.

CloudAMQP Queue

The single CloudAMQP queue which is used is named "pdfcombiner". The queue is created automatically if it doesn't exist when requested by either the Heroku Web App or the Heroku Worker. The queue is declared (i.e., requested) as non-durable, meaning that if CloudAMQP needs to restart the AMQP broker (i.e., server) which hosts the queue, any unconsumed messages on the queue will be lost. This could happen during a maintenance window when CloudAMQP needs to bounce all affected brokers.

The queue declaration is done on the following duplicated line in the code in both PDFCombinerApp/app/controllers/Application.java and PDFCombinerApp/modules/pdfworker/src/main/java/com/appirio/workers/pdf/WorkerProcess.java:

- channel.queueDeclare(QUEUE_NAME, false, false, false, null);

where QUEUE_NAME is defined as "pdfcombiner" and the first *false* parameter is for the durable flag.

Heroku Web App Overview

The Heroku Web App exposes a RESTful upload endpoint to the Salesforce app:

- POST /upload

This endpoint along with the others provided for developer or administrator use are defined in the usual place in a Play app, namely: PDFCombinerApp/conf/routes

POST /upload

As mapped in the routes file, the *upload* endpoint is handled by controller action *upload* in PDFCombinerApp/app/controllers/Application.java. You can see the logic there is simple:

1. Deserialize the request body posted by Salesforce to a JSON object (this also validates the body is valid JSON).
2. Map the JSON object to an instance of the PDFCombinerArguments class.
3. Serialize the PDFCombinerArguments instance to a binary representation (array of bytes).
4. Publish a message on the pdfcombiner queue with the array of bytes as the payload.

Note the Web App creates a new connection to the AMQP broker on receipt of the upload request and closes the connection after publishing to the queue. Also the Web App can accept simultaneous incoming requests and process them concurrently, meaning that there may be more than one AMQP connection allocated at the same time.

GET /pdf/*file

Heroku Worker Overview

Message Loop

The Heroku Worker runs a loop on startup where in each loop iteration it blocks, waiting for a message on the queue. When a message is received, the Worker removes it off the queue and processes it, after which it goes back to waiting for another message. Note that because the Worker is single-threaded with a single loop, messages in the queue are processed sequentially.

Also note that the Worker creates a new connection to the AMQP broker before entering the message loop. This single connection is used for all interaction with the queue and is never explicitly closed by the Worker, since the message loop is an infinite loop. The loop is exited (aborted) only when the Worker process terminates, after which it's up to CloudAMQP to close the connection after a timeout.

The logic for the *while(true)* loop is in PDFCombinerApp/modules/pdfworker/src/main/java/com/appirio/workers/pdf/WorkerProcess.java.

Processing of a message consists of deserializing the body (i.e., byte array payload) to an instance of the PDFCombinerArguments class and performing actions based on the properties of the object, described below.

PDFCombinerArguments (message payload)

PDFCombinerArguments.java represents the request payload sent from Salesforce to the Heroku Web App on upload, as well as the payload of the message subsequently published by the Web App to the queue and consumed asynchronously by the Heroku Worker.

The PDFCombinerArguments class is contained in PDFCombinerApp/lib/PDFCombiner.jar and is a simple POJO with these properties:

- boolean exportToPdf
- boolean exportToXls
- String attachmentsUrl
- String sessionId
- String title
- String subTitle
- String outputFileName
- boolean showTableOfContents
- boolean showPageNumbering
- boolean showTimeAndDateStamp
- boolean showTotalProgramSummary
- boolean showIndividualMarketSummary
- boolean showIndividualFlightSummary
- boolean showCoverPage
- String versionNumber
- String dateTimeStamp
- String email
- List<**PDFCombinerFile**> contents
- List<**PDFCombinerFile**> appendixes
- List<**PDFCombinerContentEntry**> pdfCombinerContentEntryList
- String clientCompanyName
- String clientContactInformation
- String agencyName
- String agencyContactInformation
- String marketName
- String marketContactInformation
- String insertContentVersionUrl

- String contentDocumentId
- boolean includeOutdoorVocabularyTermsDoc
- boolean includeResearchToolsDoc
- boolean includeServiceGuaranteeDoc
- boolean includeProductionSpecificationDoc
- String disclaimerUrl
- String shippingInstructionsUrl
- String mapPanelOrderPrefUrl
- **PDFCombinerCallback** pdfCombinerCallback
- boolean excludeFlightLines

It is expected that the Salesforce app provides JSON corresponding to the PDFCombinerArguments, although not all properties are required, with some being required if the associated boolean property is set.

You can see PDFCombinerArguments has nested objects which are bolded:
- PDFCombinerFile
- PDFCombinerContentEntry
- PDFCombinerCallback

The Salesforce app likewise provides the nested JSON corresponding to these nested objects, which are described next.

PDFCombinerFile

The PDFCombinerFile class is a simple POJO with these properties:

- String fileName
- String pathOnClient
- String salesforceUrl
- String title
- String description
- boolean showTitle
- int numberOfPages
- int startPageNumber
- boolean showAppendixEntry
- String tableOfContentsRawText
- boolean isProposalReport
- List<String> fieldNamesPipeDelimited
- List<String> fieldLabelsPipeDelimited
- List<String> fieldTypesPipeDelimited
- List<String> fieldTotalsPipeDelimited
- String buyType

PDFCombinerContentEntry

The PDFCombinerContentEntry class is a simple POJO with these properties:

- String title
- String description
- int pageNumber

PDFCombinerCallback

The PDFCombinerCallback class is a simple POJO with these properties:

- String callbackUrl
- String callbackContents

Sample JSON representation

For example, to represent PDFCombinerCallback, the Salesforce app would fill out JSON like this nested in the request body:

```
{
  "callbackUrl":"https://na14.salesforce.com/services/data/v27.0/sobjects/Account/001d000000HiIUz",
  "callbackContents":"{\"Site\" : \"%result%\"}"
}
```

Heroku Worker Flow

On receipt of a message and deserialization of the binary payload to an instance of the PDFCombinerArguments class, the Heroku Worker proceeds as follows:

Download the disclaimer file

Download the disclaimer file specified by the disclaimerUrl property in PDFCombinerArguments from Salesforce to local path /tmp/disclaimers.xml (on Heroku).

Recall that because the Worker is single-threaded, there is no concern of overwriting an existing disclaimers.xml in the /tmp directory on the Heroku worker dyno.

Also note that all download requests to Salesforce (including the ones described below) are authenticated via OAuth using the sessionId property in PDFCombinerArguments, as seen in the following line from the downloadFile method of

PDFCombinerApp/modules/pdfworker/src/main/java/com/appirio/workers/pdf/WorkerProcess.java:

- uCon.setRequestProperty("Authorization", "Bearer " + sessionId);

All files (including those described below) are downloaded using a 1K buffer for each read from the input stream, to avoid excessive memory usage.

Download the shipping instructions file (if exportToPdf)

If the exportToPdf flag is set in PDFCombinerArguments, download the shipping instructions file specified by the shippingInstructionsUrl property in PDFCombinerArguments from Salesforce to local path /tmp/shippingInstructions{*uniqueId*}.xml, where *{uniqueId}* is the current timestamp in milliseconds.

Download the map panel order preferences file

Download the map panel order preferences file specified by the mapPanelOrderPrefUrl property in PDFCombinerArguments from Salesforce to local path /tmp/ mapPanelOrderPref{*uniqueId*}.xml

Download the content file(s) to be combined in the output PDF

If the PDFCombinerArguments input has a list of contents (one or more PDFCombinerFile(s)), download each file as specified by the salesforceUrl property in the PDFCombinerFile from Salesforce to local path /tmp/*{fileNameOnClient}*, where *{fileNameOnClient}* is specified by pathOnClient property in the PDFCombinerFile.

Special meaning is given to a downloaded file which is XML. In this case, the file is not actually one which is combined as-is in the output PDF. Instead, an XML file represents data from Salesforce (a QueryResult) for use in generating a report to be combined in the output PDF along with the other downloaded content files.

The generation of the report to be combined is described in the **Report Generation** section below.

Download the appendix file(s)

If the PDFCombinerArguments input has a list of appendixes (one or more PDFCombinerFile(s)), download each file as specified by the salesforceUrl property in the PDFCombinerFile from Salesforce to local path /tmp/*{fileNameOnClient}*, where *{fileNameOnClient}* is specified by pathOnClient property in the PDFCombinerFile.

Combine downloaded and generated files into the output PDF

Content file(s), generated report(s), and appendix(es)

An instance of the PDFCombiner class is created and passed the original PDFCombinerArguments updated with the local paths of the following files to be combined into a single output PDF:

- the content file(s) and generated report(s) - if the excludeFlightLines flag is not set in PDFCombinerArguments
- the appendix file(s)

Any content or appendix files which are not PDF are converted to PDF before combining. Supported formats are:
- doc, docx
- xls, xlsx
- ppt
- images

Generated cover page and table of contents

Additionally, the following generated file(s) are included in the output PDF:
- a cover page PDF - if the showCoverPage flag is set in PDFCombinerArguments
- a table of contents PDF - if the tableOfContents flag is set in PDFCombinerArguments

These files are generated from templates with properties from PDFCombinerArguments used to replace placeholders or populate sections in the template.

<u>For the cover page</u>, the template is coverpage.pdf in the directory specified by the PDFS_TEMPLATES_DIR environment variable, with these properties from PDFCombinerArguments used to replace corresponding placeholders in the template:
- title
- subTitle
- clientCompanyName
- clientContactInformation
- agencyName
- agencyContactInformation
- marketName
- marketContactInformation
- versionNumber

For the table of contents, the template is table_of_contents.pdf also in the directory specified by the PDFS_TEMPLATES_DIR environment variable, with properties and contents from PDFCombinerArguments used to populate sections in the template, in particular the list of PDFCombinerContentEntry which has title, description, and page number properties for each content entry.

Both the generated cover page and table of contents are created as randomly named files in the system temp directory with prefix "temp_" and extension ".pdf".

Output PDF

The output PDF file is created as {*uniqueId*}.pdf in the directory specified by environment variable GENERATED_PDFS_DIR, where *{uniqueId}* is the current timestamp in milliseconds.

During merging of various files into a single PDF, these properties of PDFCombinerArguments are used for control of content or further content in the output:

- showPageNumbering
- marketContactInformation
- clientContactInformation
- dateTimeStamp - if showTimeAndDateStamp is set

Additionally for each file combined, these properties from the associated PDFCombinerFile entry are used likewise for control of content for that file in the output:

- isProposalReport
- title - if showTitle is set

Upload Output PDF to Salesforce

Finally upload the combined output PDF to Salesforce as multipart form data consisting of two parameters:

- **entity_content** - with value as a serialized JSON object, content type *application/json*
- **VersionData** - with value as the binary content of the output PDF, content type *application/octect-stream* (note the typo is not mine, it's in the source code PDFCombinerApp/modules/pdfworker/src/main/java/com/appirio/workers/pdf/FileUploader.java)

The entity_content value is the following JSON with placeholder values {...} replaced by corresponding properties from PDFCombinerArguments as passed from Salesforce in the original (incoming) upload request:

```
{
  "Title": {title},
  "Description": {subTitle},
  "FirstPublishLocationId": {contentDocumentId},
  "ReasonForChange": {title},
  "PathOnClient": {outputFileName}.pdf
}
```

This outgoing multipart form data upload to Salesforce is posted to the URL specified by the insertContentVersionUrl property in the PDFCombinerArguments, and is authenticated via OAuth using the sessionId property in PDFCombinerArguments.

Notify the PDF Combiner Callback URL

Salesforce responds to the upload with a JSON payload containing an id property, actually the URL of the PDF output file stored in Salesforce. This "id" value is parsed out of the response and used to make a callback to the Salesforce URL specified by the PDFCombinerCallback.callbackUrl property of the PDFCombinerArguments.

The body of the response for the callback URL is generated from a template contained in the PDFCombinerCallback.callbackContents property of the PDFCombinerArguments. The template contains the following placeholders which are replaced as specified:
%PDFOutput% - replaced with the URL of the PDF output file stored in Salesforce
%Status% - replaced with "Completed"

This completes the Worker flow which initiated with an upload request posted to the Heroku Web App from Salesforce.

Exception Handling

Note that if there is any non-fatal exception in the flow the Worker still calls the callback URL, but with %Status% set to "Failed", and the Worker continues to the next iteration in the message loop, waiting for another message.

Report Generation

PDF generation from Salesforce data as XML

As mentioned previously, special meaning is given to a downloaded content file which is XML. In this case, the file is not actually one which is combined as-is in the output PDF. Instead, an XML file represents data from Salesforce (a QueryResult) for use in generating a report to be combined in the output PDF along with the other downloaded content files.

First the XML file is downloaded from Salesforce (authenticated as with all file downloads) and saved in the /tmp directory with the name specified in the pathOnClient property of the corresponding PDFCombinerFile entry.

An instance of the Reporter class is created and used to generate a PDF report based on the data in the XML file, along with the contents of these already downloaded files:

- the disclaimer file
- the shipping instructions file
- the map panel order preferences file
- the appendix file(s)

and also using these boolean properties from the incoming PDFCombinerArguments:

- showTotalProgramSummary
- showIndividualMarketSummary
- showIndividualFlightSummary
- exportToPdf

as well as using the values of these properties from the corresponding PDFCombinerFile entry:

- fieldNamesPipeDelimited
- fieldLabelsPipeDelimited
- fieldTypesPipeDelimited

The Reporter (located in PDFCombinerApp/lib/ReportGenerator.jar) generates the PDF report using the JasperReportBuilder class provided by the DynamicReports library. The report is created as {*uniqueId*}.pdf in the directory specified by environment variable GENERATED_PDFS_DIR, where *{uniqueId}* is the current timestamp in milliseconds.

The Reporter also generates a CSV file with the table of contents for the report, with file name as the same *{uniqueId}* and extension as *.csv*.

The Worker takes the output from the Reporter and includes the path to the generated PDF report as one of the files to be combined into the final output PDF.

The Worker also uses the CSV output from the Reporter to add table of contents information to the PDFCombinerArguments as an entry in the PdfCombinerContentEntryList.

Excel export option

If the exportToXls flag is set in the incoming PDFCombinerArguments, the Reporter additionally generates an Excel version of the report. The Excel report is created as {*uniqueId*}.xls, likewise in the directory specified by environment variable GENERATED_PDFS_DIR.

In the case of exportToXls being set, the generated Excel report is uploaded to Salesforce by the Worker immediately, and not included in the output PDF combining. The upload is done as described previously in the **Upload Output PDF to Salesforce** section, except that the PathOnClient property in the JSON data ends in *.xls* instead of *.pdf*.

After the upload, the Worker calls the PDF combiner callback URL as previously described in the **Notify the PDF Combiner Callback URL** section, except that the %ExcelOutput% placeholder is replaced instead of the %PDFOutput% one with the URL of the file on Salesforce.

Running on Heroku

Deployment

Here are the 4 steps to deploy the OutputBuilder (pdfcombinercco) service, consisting of a Web App and a Worker, to Heroku.

1. Clone the repo from GitHub:

- git clone https://github.com/cloudspokes/pdfcombinercco

2. Cd to the newly created pdfcombinercco directory and create a new Heroku app from there:

- heroku create

3. Add the CloudAMQP addon for Heroku to your new app, choosing *lemur* for the free edition:

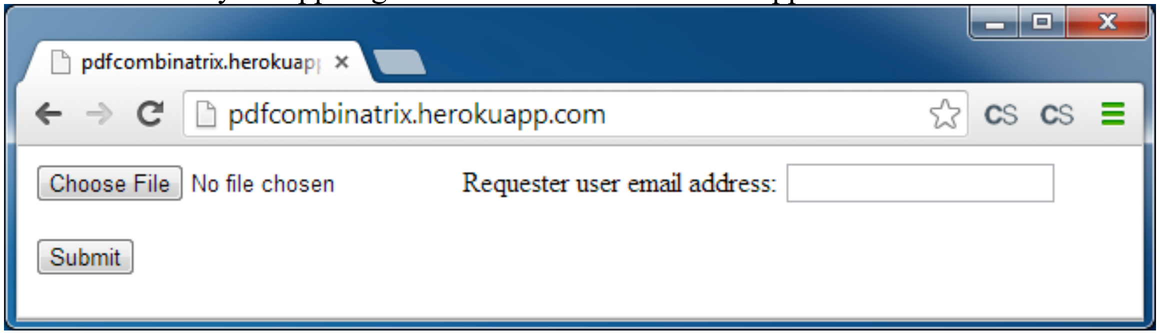
- heroku addons:add cloudamqp:lemur

4. Push the code to Heroku:

- git push heroku master

This may take a while considering that you need to upload the jars in the lib and modules directories, but that's it, you're done for the deployment step.

Hit the URL for your app to get the test form for the Web App:



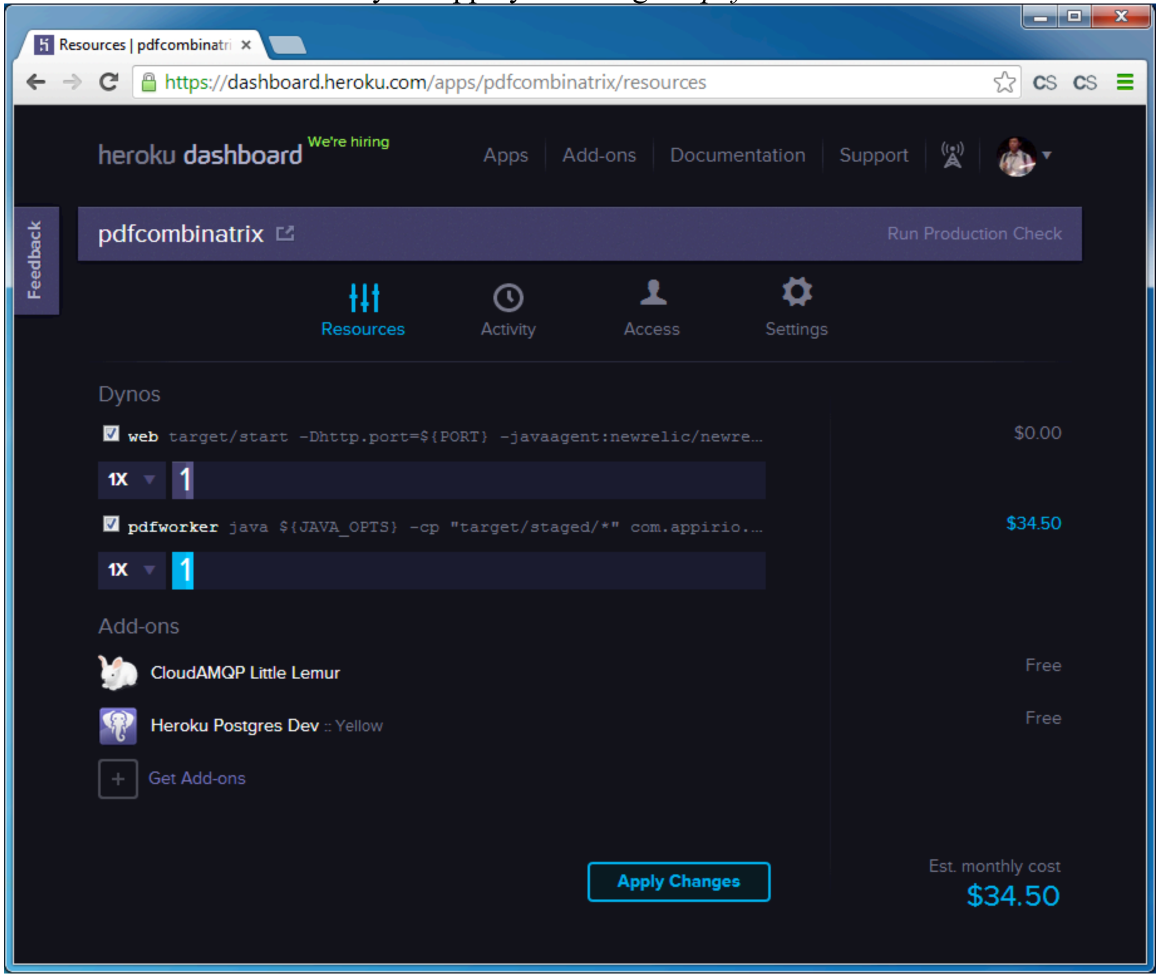
Scaling the worker dyno

By default the web dyno is scaled and running after your deployment, meaning the Web App is running.

However, the worker dyno is not scaled, meaning the Worker is not running (and will not run), resulting in any messages produced by the Web App not be processed (consumed by the Worker). Do the scaling either from the command line:

- heroku ps:scale worker=1

or from the resources UI for your app by checking the *pdfworker* checkbox as shown:



Dynos

☑ web target/start -Dhttp.port=${PORT} -javaagent:newrelic/newre… $0.00
1X 1

☑ pdfworker java ${JAVA_OPTS} -cp "target/staged/*" com.appirio… $34.50
1X 1

Add-ons

CloudAMQP Little Lemur Free

Heroku Postgres Dev :: Yellow Free

Get Add-ons

Est. monthly cost $34.50

Note the cost beside the *Apply Changes* button if you intend to leave the worker dyno running long-term.

Configuring the app

The following 3 environment variables are used by the app:

- CLOUDAMQP_URL

- PDFS_TEMPLATES_DIR
- GENERATED_PDFS_DIR

CLOUDAMQP_URL

This is already set by the cloudamqp addon, for example

- heroku config:get CLOUDAMQP_URL

returns
amqp://umbhbjyw:gcgKHqrLtjAOJ5dLqhsUb_pLHrel1gpz@turtle.rmq.cloudamqp.com/umbhbjyw

PDFS_TEMPLATES_DIR

This is the location of the existing PDF template files (coverpage.pdf and table_of_contents.pdf) which are part of the deployed app. Add it with:

- heroku config:add PDFS_TEMPLATES_DIR=/app/public/pdfs

GENERATED_PDFS_DIR

This is the location where PDF files generated by the app at runtime are stored. Add it with:

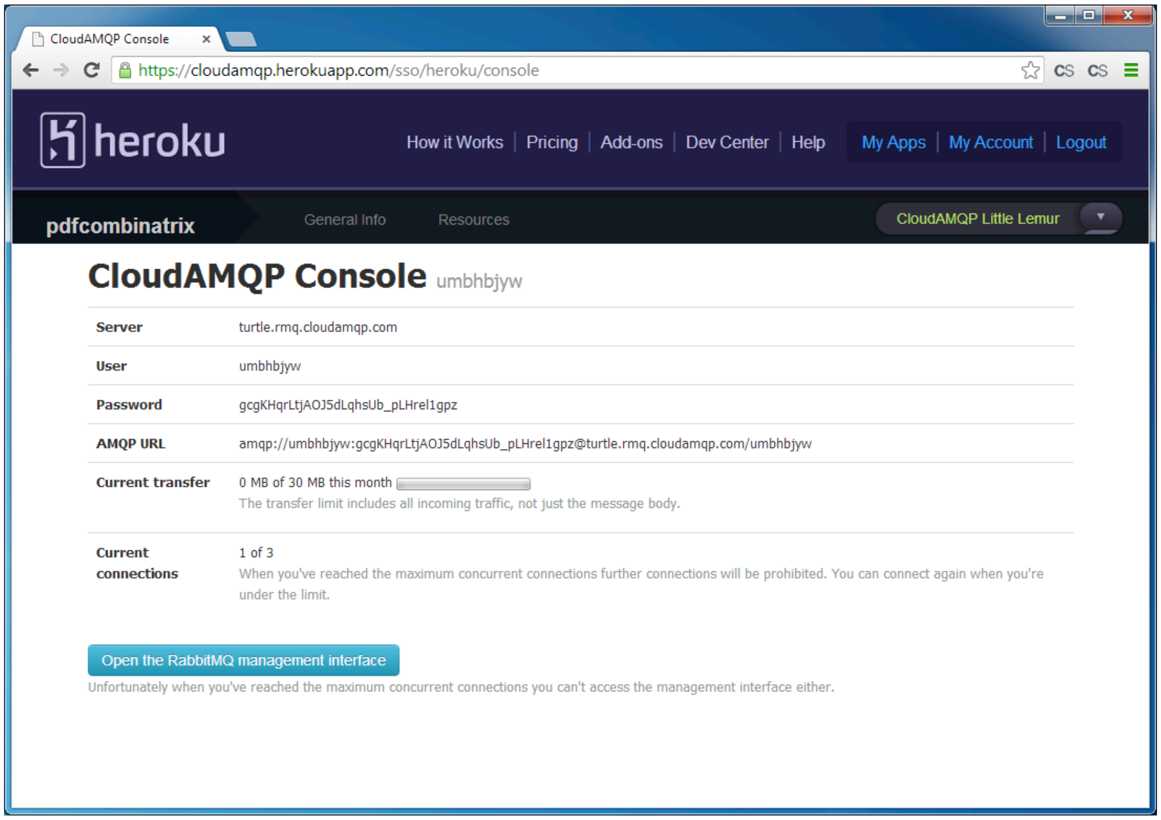
- heroku config:add PDFS_TEMPLATES_DIR=/app/public/pdfs

Note that all generated PDF files will be lost when the app is restarted by Heroku.

The app also creates files in the system temp directory (/tmp). These likewise will be lost when the app is restarted.

Queue Monitoring

Click on the cloudamqp addon link in the resources UI for your app to see:


CloudAMQP Console
https://cloudamqp.herokuapp.com/sso/heroku/console
heroku
How it Works | Pricing | Add-ons | Dev Center | Help
My Apps | My Account | Logout
pdfcombinatrix
General Info
Resources
CloudAMQP Little Lemur

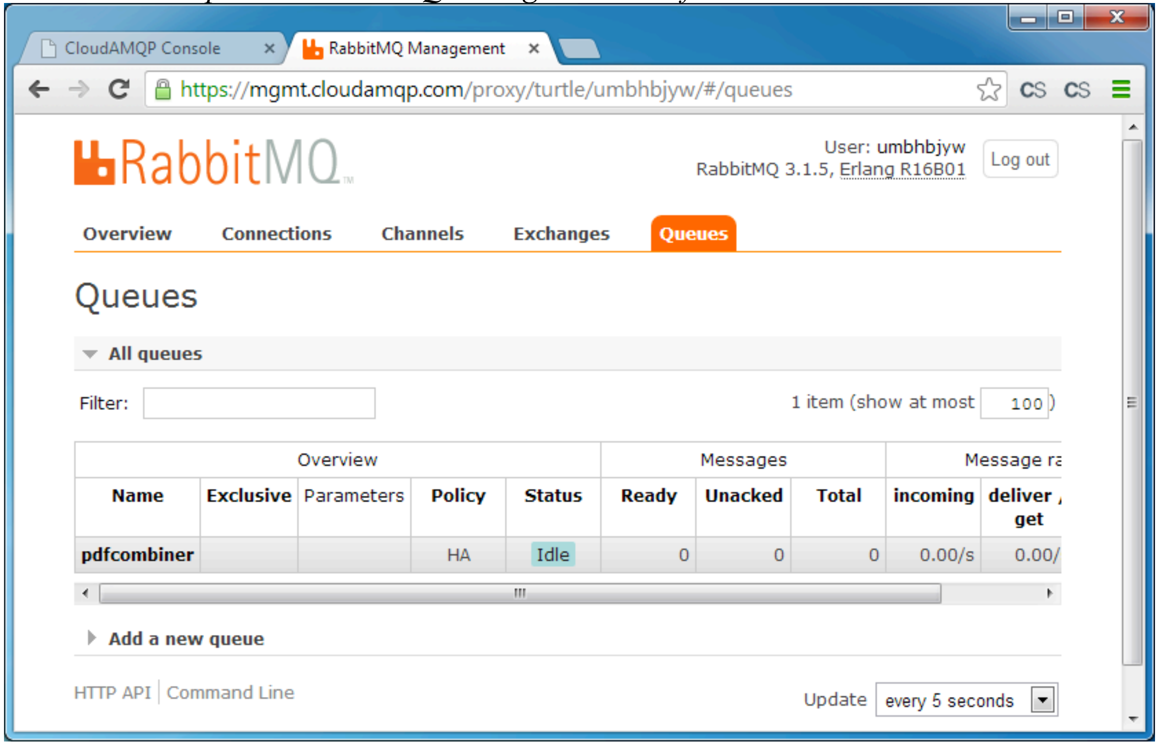
CloudAMQP Console umbhbjyw

Server	turtle.rmq.cloudamqp.com
User	umbhbjyw
Password	gcgKHqrLtjAOJ5dLqhsUb_pLHrel1gpz
AMQP URL	amqp://umbhbjyw:gcgKHqrLtjAOJ5dLqhsUb_pLHrel1gpz@turtle.rmq.cloudamqp.com/umbhbjyw
Current transfer	0 MB of 30 MB this month The transfer limit includes all incoming traffic, not just the message body.
Current connections	1 of 3 When you've reached the maximum concurrent connections further connections will be prohibited. You can connect again when you're under the limit.

Unfortunately when you've reached the maximum concurrent connections you can't access the management interface either.

Note that you should have at least 1 current connection which was created by the Worker on startup. This number will increase as the Web App receives requests from Salesforce.

Click on blue *Open the RabbitMQ management interface* button to see:



Overview					Messages			Message ra	
Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	incoming	deliver / get
pdfcombiner			HA	Idle	0	0	0	0.00/s	0.00/

There's the *pdfcombiner* queue which was automatically created by either the Worker or the Web App, whichever tried to access the queue first.

Running locally

Prerequisites - Play

It is assumed you are a *skilled Play developer*. Therefore you should already have Java installed, at least JDK 1.6, and a version of Play installed. However the version of Play used by the app is version 2.04 which you may not have since the latest version is 2.2.0.

Therefore to avoid compatibility issues, download version 2.04 from http://downloads.typesafe.com/releases/play-2.0.4.zip and configure your environment if need be for this version.

After downloading and unzipping the Play 2.0.4 install, cd to the pdfcombinercco directory and run *play* (assuming you've added the command from the Play install directory to your path). As you know this launches the Play shell. Then simply enter *run* in the shell as usual to start the Web App:

```
[PDFCombinerApp] $ run

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)
```

Hit the app at http://localhost:9000

Setting up Rabbit MQ

If you don't already have Rabbit MQ installed, download it from http://www.rabbitmq.com/download.html for your operating system, and set up the AMQP broker locally.

Configuring the app

As with running on Heroku, the following 3 environment variables are used by the app:

- CLOUDAMQP_URL
- PDFS_TEMPLATES_DIR
- GENERATED_PDFS_DIR

You can leave CLOUDAMQP_URL unset if you want because the app will connect to amqp://guest:guest@localhost (your local broker endpoint) if the environment variable isn't set.

For PDFS_TEMPLATES_DIR set it to the full path of your PDFCombinerApp/public/pdfs directory.

For GENERATED_PDFS_DIR I suggest setting it to a directory outside your project, to avoid the generated files from being included in git commits you do from the pdfcombinercco project.

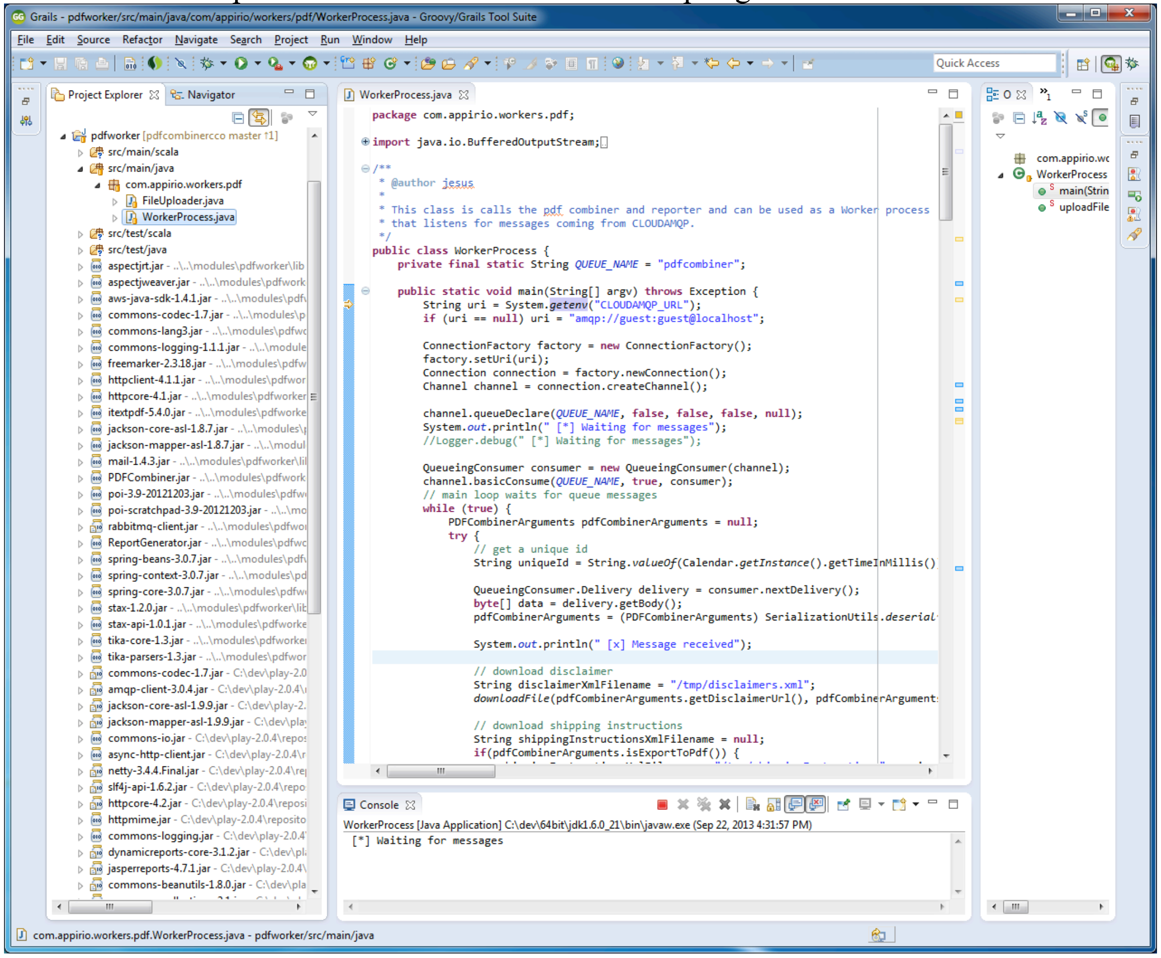
After setting the environment variables, you need to restart the app in a shell which has the new environment variables loaded.

Running the Worker

The Worker is a separate process independent of the Web App. In fact, the Worker is not a Play app and has no dependencies on Play even though its code is contained within the Play app project, in directory:

- PDFCombinerApp/modules/pdfworker

The easiest way to run the Worker is to import the code in the pdfworker directory into Eclipse (or another IDE), so that you can run it as a regular Java app, as I did using my IDE which is Eclipse-based Grails Tool Suite from SpringSource:


Grails - pdfworker/src/main/java/com/appirio/workers/pdf/WorkerProcess.java - Groovy/Grails Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Project Explorer
Navigator
WorkerProcess.java
Console

```
package com.appirio.workers.pdf;

import java.io.BufferedOutputStream;

/**
 * @author jesus
 *
 * This class is calls the pdf combiner and reporter and can be used as a Worker process
 * that listens for messages coming from CLOUDAMQP.
 */
public class WorkerProcess {
    private final static String QUEUE_NAME = "pdfcombiner";

    public static void main(String[] argv) throws Exception {
        String uri = System.getenv("CLOUDAMQP_URL");
        if (uri == null) uri = "amqp://guest:guest@localhost";

        ConnectionFactory factory = new ConnectionFactory();
        factory.setUri(uri);
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        System.out.println(" [*] Waiting for messages");
        //Logger.debug(" [*] Waiting for messages");

        QueueingConsumer consumer = new QueueingConsumer(channel);
        channel.basicConsume(QUEUE_NAME, true, consumer);
        // main loop waits for queue messages
        while (true) {
            PDFCombinerArguments pdfCombinerArguments = null;
            try {
                // get a unique id
                String uniqueId = String.valueOf(Calendar.getInstance().getTimeInMillis()

                QueueingConsumer.Delivery delivery = consumer.nextDelivery();
                byte[] data = delivery.getBody();
                pdfCombinerArguments = (PDFCombinerArguments) SerializationUtils.deserial

                System.out.println(" [x] Message received");

                // download disclaimer
                String disclaimerXmlFilename = "/tmp/disclaimers.xml";
                downloadFile(pdfCombinerArguments.getDisclaimerUrl(), pdfCombinerArgument

                // download shipping instructions
                String shippingInstructionsXmlFilename = null;
                if(pdfCombinerArguments.isExportToPdf()) {
```

```
WorkerProcess [Java Application] C:\dev\64bit\jdk1.6.0_21\bin\javaw.exe (Sep 22, 2013 4:31:57 PM)
 [*] Waiting for messages
```

To import the pdfworker code into Eclipse, as usual you run the *play eclipsify* command to create an Eclipse project (with.project and .classpath files, etc. as you are familiar with) which you can easily import into Eclipse. Even though pdfworker under modules is not a Play app, *play eclipsify* will still generate an importable Eclipse project.

.gitignore

Note that if you use an IDE to run the app, you should define a .gitignore file in the project which ignores IDE-specific files you want to exclude on git commits.

For Eclipse, use this as the contents of your .gitignore file:

```
.classpath
.project
.settings/org.scala-ide.sdt.core.prefs
```

```
.target/
logs/
modules/pdfworker/.classpath
modules/pdfworker/.project
modules/pdfworker/target/
project/project/
project/target/
target/
```

Now you are ready to make changes to this wonderful tool!

Enjoy,
William Cheung
First Day of Autumn 2013 (Sunday September 22)
Toronto